Matt Bradford, Alex Craig
6.111: Introduction to Digital Design
12/10/2020

Sound Localization Project Report

1 Introduction

Locating the source of a sound has many applications. The sensor company FLIR has an industrial “acoustic camera” with 124 microphones designed to help locate mechanical problems by finding the source of the sound they create. Ships and submarines use sonar to navigate. Humans also use interaural time delay to determine the direction of arrival of sound. The goal of our project was to locate sound sources by calculating the time delay of arrival of sound at multiple microphones.

1.1 Using Delay to Determine Source Location

The simplest way to get location information from sound uses 2 microphones as shown in figure 1 below. Call

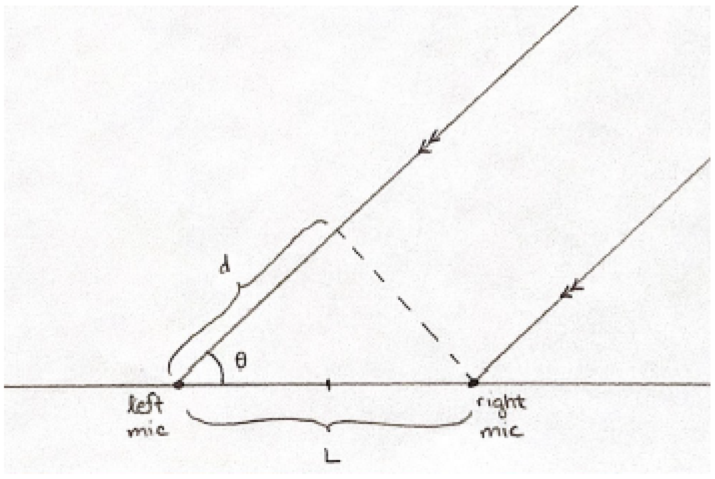

Figure 1: A planar diagram of a 2-microphone system.

the distance between the microphones L. We will always assume that the sound source is far enough away from the microphones that we can approximate the sound rays as coming in parallel to our 2 microphones (we will call this the far-source approximation). With this assumption, the sound has to travel a distance $d = L\cos\theta$ (where θ is the angle of arrival measured against the axis between the microphones) farther to reach the left microphone than to reach the right microphone. This will introduce a time delay

$$\Delta t = \frac{d}{c} = \frac{L\cos\theta}{c}$$

(where $c = 343$ m/s is the speed of sound in air) in the left audio signal. In a digital system we will be sampling the audio from both mics at some rate f_s, and the main goal of the system will be to determine the delay in terms of number of audio samples; call this delay k. Then the actual time delay is

$$\Delta t = \frac{k}{f_s}$$

and so we can find the cosine of the angle of arrival as

$$\cos\theta = k \cdot \frac{c}{f_s \cdot L}$$

If $k = 0$, then the signal arrived at both mics at the same time: this means that the angle of arrival is $\pi/2$, or the source is directly in front of the mics. If the number of samples delay is negative ($k < 0$), this means that the signal arrives at the left mic first, so the angle of arrival is greater than $\pi/2$.

The term

$$\frac{c}{f_s \cdot L}$$

is critical because it describes the smallest change in $\cos\theta$ that we can measure with our device; it basically describes the resolution of our system. c/f_s is the distance sound travels in a sample period. L, the distance between the mics, is the aperture of our system. The resolution is the ratio of the distance sound travels in a sample period to the aperture. We can improve resolution (*i.e.,* making this smaller) by increasing or sample rate our increasing our aperture. For our system our sampling rate was limited by the microphones; the problem with increasing aperture is that it increases the source range required to satisfy our far-source assumption. Interestingly, increasing c (for example, underwater) decreases resolution, which would be a is for sonar arrays.

The planar diagram can be a little misleading. If we know the angle of arrival θ, this only narrows the location of the source down to somewhere on a cone whose axis of symmetry is the line through the mics and whose vertex angle is 2θ (see figure 2 for an example of such a cone). Under the far-source approximation all such points on the cone will result in exactly the same delay between the audio signals, and thus are indistinguishable to a simple 2-mic system. Importantly, we can't even approximate the range to the source using just 2 mics. We call this kind of localization "1-D localization" because we determine only a single location parameter θ.

Although we weren't able to get this far in our project, you could get more location information with 3 microphones. With a mic at the origin, another on the x-axis, and another on the y-axis you can calculate angles α and β with respect to the x- and y-axes, which is enough to narrow the source location down to a ray emanating from the origin (again making the far-source approximation). We call this "2-D localization" because we can compute 2 location parameters.

To compute the range (the parameter we're missing with the 3 microphone set-up above), we need to introduce another set of microphones relatively far away. This is confusing because it seems like the 3 delays available between 3 microphones should be enough to determine 3 location parameters. There are at least 2 problems with this logic. First, the 3 delays from 3 microphones are not actually independent (*i.e.,* if we know the delay between mics A and B and between mics B and C then we also know the delay between mics A and B); thus we're only truly measuring 2 independent variables, so we can only really expect to be able to compute 2 independent parameters. Second, the far-source approximation means we're assuming that the distance between our mics is much smaller than the distance to the source; this is a fundamental barrier to calculating the range to the source accurately (because we just don't have enough breadth of perspective; this is why you can't get depth perception with a single eye). This is why we need a separate array in a different position to get information about the range.

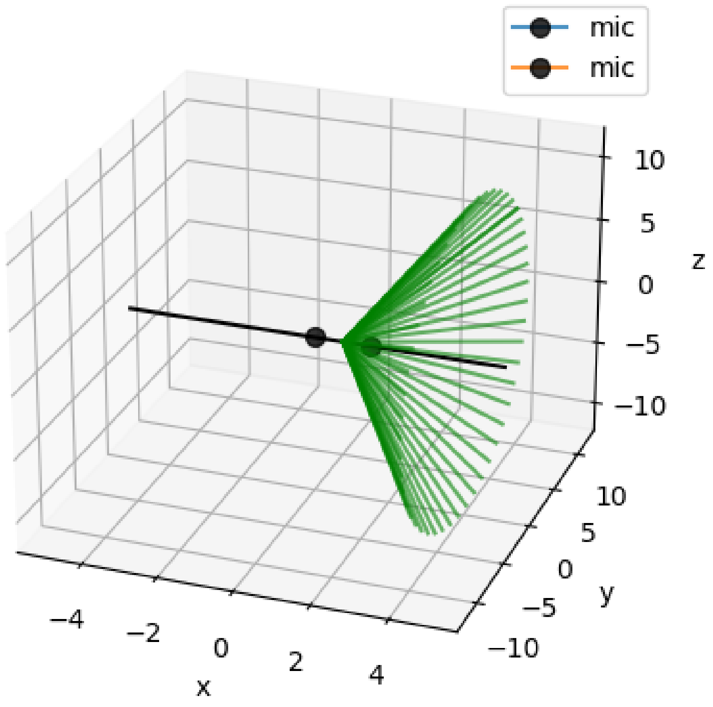

Figure 2: A cone showing the possible locations of the source for a particular θ (in this case approximately $\theta = \frac{\pi}{4}$).

1.2 The Mechanics of the Jeffress Circuit

In preparation for this project, we did a bit of research into what people had done previously[1][2][3]. Most of the implementations I found required GPU or even CPU architecture implementations to do all of the math. The attempts on FPGAs were all running at very high clock speeds with a lot of microphones. The source of the first set of implementation inefficiencies were discovered by observing that the GPU/CPU implementations were calculating the Generalized Cross-Correlation (GCC) of the audio signals from different microphones. The second set is due to snarled computation in which massive and therefore acoustically complex microphone arrays are used for redundant operations in which efficiency is only gained by running at a high speed. The Jeffress circuit solves these issues.

Lloyd Alexander Jeffress, an American Psychoacoustic scientist, first wrote about the circuit in his 1948 paper *A place theory of sound localization*[4]. He proposed that humans determined interaural time delays by the use of place coding. This involves sampling a neural signal at various points along its neuronal membrane. The Jeffress circuit essentially detects which signal wins a race to it and by how much. The 'call' is made by coincidence detectors that are at set points along each signal path. The race ends when the two signals trip a latch. Because the signals are coming from opposite sides of the head, these signal lines are anti-parallel. This yields a fair race condition in which a tie would be meeting at the center of the observed signal paths. It was later found that such a circuit does exist in the Medial Superior Olive of the human brain[5].

Further research found a paper from 2016 by Kugler et al. that dealt with this same task and implemented Jeffress time delay extraction on an FPGA [6]. Additional modifications were made to their system, notably the lack of a clock or RAM usage in our Jeffress implementation.

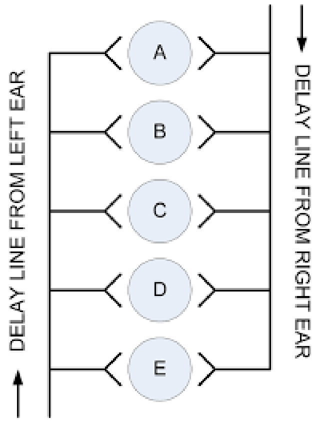

Figure 3: Jeffress circuit layout. A-E are coincidence detectors.

2 Modules

2.1 I2S Interpreter Module (Alex)

The microphones we used give digital output using the I2S protocol. I2S uses 3 main lines:

1. Bit clock (*aka*, serial clock; labeled SCK on the mics and `i2s_sclk` in the simulation screenshot below): a periodic signal provided by the interpreter that the microphone uses to synchronize itself.

2. Word select (*aka*, left-right clock; labeled LRCL on the mics and `i2s_ws` in the simulation below): a signal that the interpreter uses to indicate which microphone should send data.

3. Data (labeled DOUT on the mics and `i2s_data` in the simulation below) a signal that is controlled alternately by both mics. It carries the audio signal bit by bit.

In addition to these 3 wires, each microphone also needs power (an additional 2 wires per microphone), and a channel ID signal (labeled SEL on the mics) that tells them whether they are the left or right mic (these signals were hard-coded in the top-level of our project because they never need to change). Our microphone set-up and wiring are shown in figure 5. Five of the 6 wires can be shared between the microphones; only the SEL wire needs to be different because it distinguishes right from left.

Figure 6 is a screenshot of a simulation of the I2S receiver module (the `i2s_data` signal here is just a test signal generated by the simulation to evaluate the receiver module):

The bit clock signal (`i2s_sclk`) runs at 4 MHz (25 times slower than the 100 MHz system clock); in real life whichever microphone was indicated by the word select signal (`i2s_ws`) would send a single bit of its

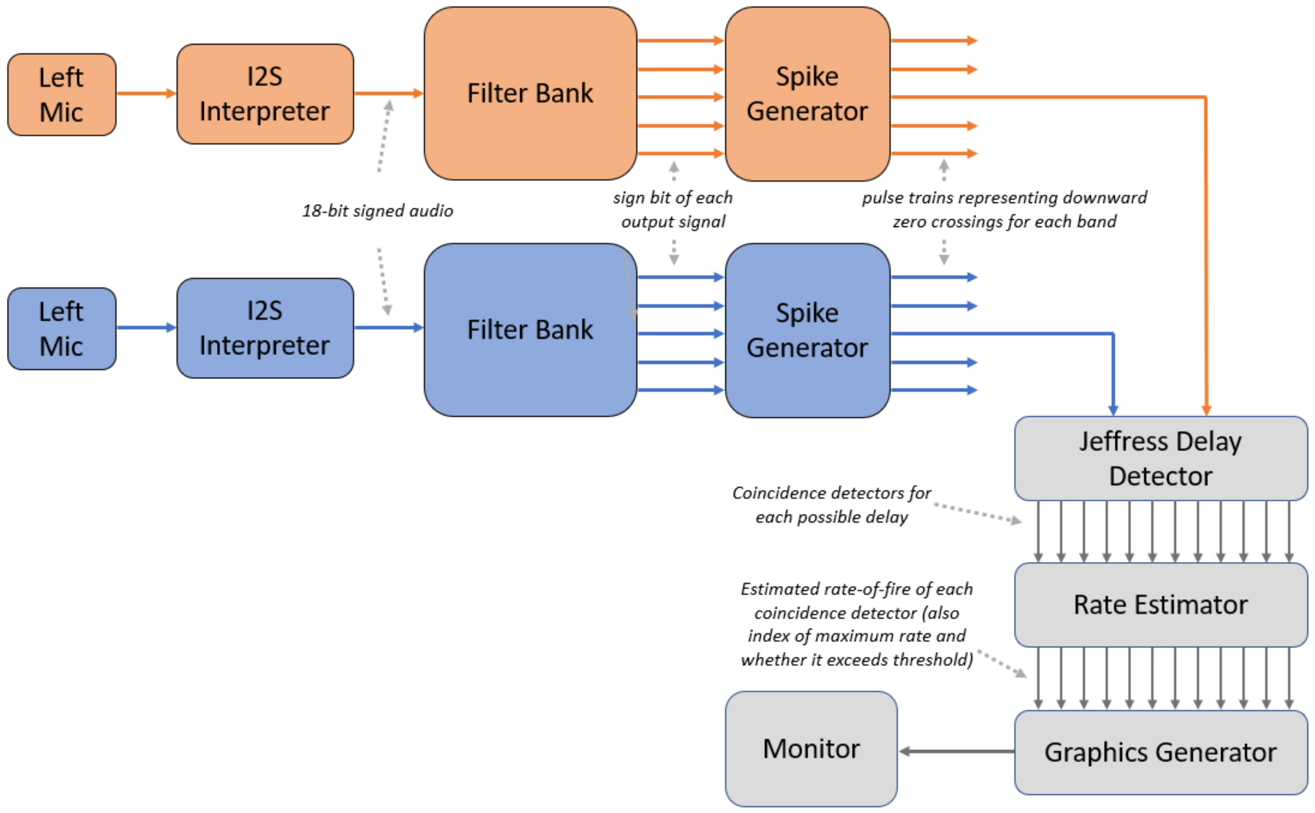
Left Mic
I2S Interpreter
Filter Bank
Spike Generator
18-bit signed audio
sign bit of each output signal
pulse trains representing downward zero crossings for each band
Left Mic
I2S Interpreter
Filter Bank
Spike Generator
Jeffress Delay Detector
Coincidence detectors for each possible delay
Rate Estimator
Estimated rate-of-fire of each coincidence detector (also index of maximum rate and whether it exceeds threshold)
Monitor
Graphics Generator

Figure 4: The block diagram for our system. The left audio pipeline is colored orange, and the right audio pipeline is colored blue.

audio signal along the data line for each period of the bit clock. The word select signal flips every 32 periods of the bit clock (I don't think this is an I2S standard, but the microphones we used require the word select period to be exactly 64 times the bit clock period; this took a little help from Gim to figure out). With this configuration each microphone sends 32 bits of audio each time it is called on; in reality, only 18 bits contain useful information.

Flipping the word select signal every 32 periods of the 4 MHz bit clock corresponds to a sample rate of 62.5 KHz for each microphone. This was the fastest the microphones were rated to run. We used the fastest sampling rate possible because it allowed us to detect the delay between left and right audio signals more precisely.

Once I figured out all the details of the protocol (especially that I had to hold the word select signal constant for 32 bit clock periods, even though we only wanted 18 bits of data), the I2S receiver wasn't too complicated. It has to generate the bit clock and word select signals at the correct frequencies. It also stores 2 18-bit values for the left and right audio signals, writes into them 1 bit at a time as it gets data from the mics, and then pulses the appropriate `valid_out` signal whenever it is finished reading an audio sample.

Because our samples from the two mics are alternating (*i.e.,* we're never hearing from both mics at exactly the same time), our delay can never be exactly 0—it's always an integer $\pm\frac{1}{2}$. This is why there are an even number of possible delays, and there is no block or arc directly in the middle of the display.

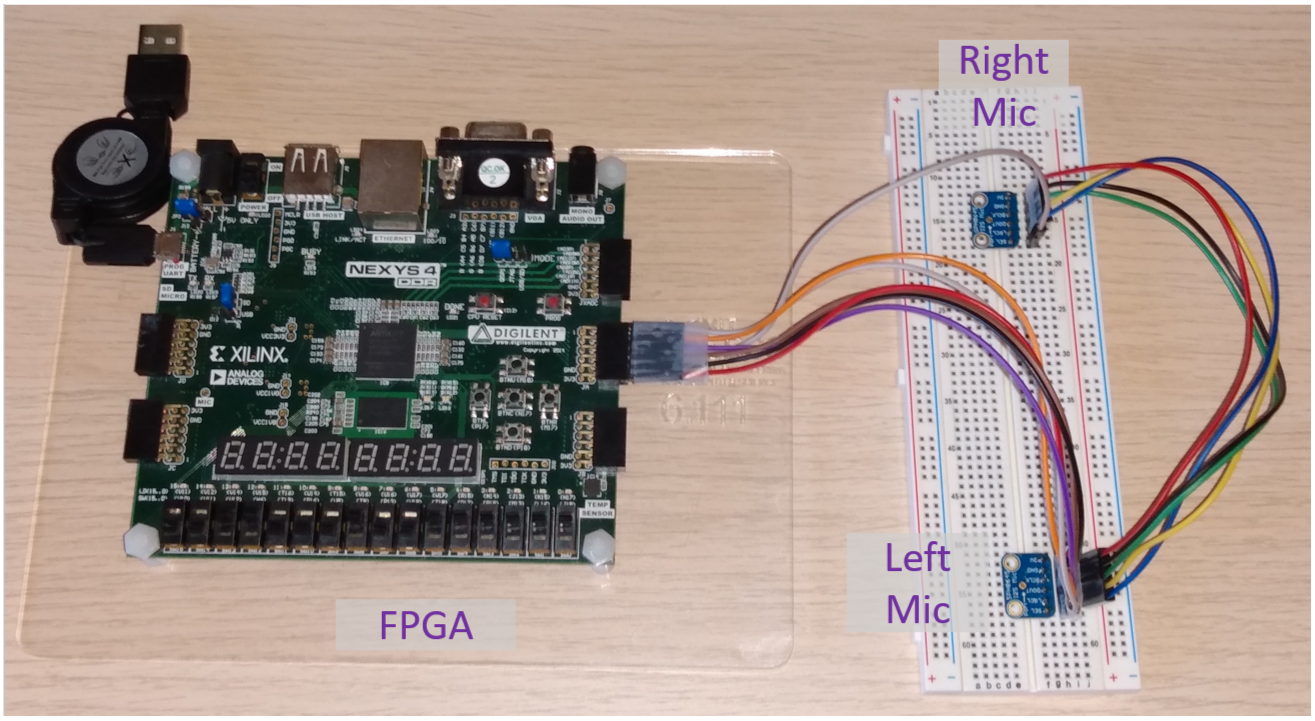

Figure 5: Our 2 microphones wired to the FPGA.

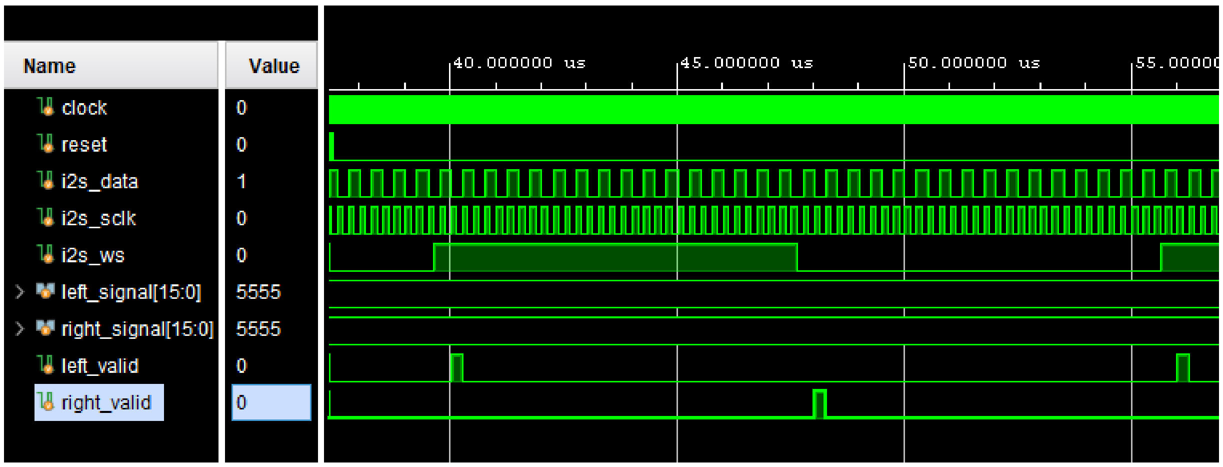

Figure 6: A screenshot from simulation of the I2S receiver module.

2.2 Filter Bank (Alex):

I probably spent most of my time designing, writing, and debugging the filter bank module. We need a filter bank because the input to the Jeffress circuit needs to be basically sinusoidal. Thus we want to split each audio signal into bands of relatively narrow frequency content. It's an interesting question how narrow your bands need to be to look like sinusoids. The Kugler paper used 64 subbands from 100 Hz to 8 KHz (which comes out to roughly 125 Hz per band); in our final demo we used only 16 bands from 100 Hz to 3 KHz (which comes out to roughly 180 Hz per band; our max frequency was lower because our mics were spaced farther apart). Our demo worked pretty well; with more time it would have been interesting to try with more channels (although resources were also a problem, as I'll explain later).

The first big question was whether to use IIR or FIR filters. I had already implemented an FIR filter in lab 5a, but the Kugler paper used IIR filters. I decided on IIR filters because in general you need many fewer coefficients to make an IIR filter with a certain cutoff magnitude compared to an FIR filter; the number of coefficients seemed important to minimize because (a) at the time we wanted to be able to synthesize 3 audio channels (for 3 microphones) $\times$ 64 subbands per channel = 192 of these filters and (b) I estimated that an FIR filter that could separate out pass-bands on the order of 100 Hz would need hundreds of coefficients (I don't know the best way to do this calculation exactly and I didn't invest much time in it). Just storing 64 subbands $\times$ 250 coefficients per subband $\times$ 32 bits per coefficient (all rough estimates) $\approx$ 64 Kilobytes seemed like a lot, so I decided to go with IIR filters.

After some conversations with my EE dad, it sounds like there is a really slick and resource-light way to implement exactly the kind of filter bank we wanted. If I had had more time to learn more about it I would really have enjoyed making something smarter, but we ended up being pressed for time, and making a slicker filter bank was not our primary concern.

Another big question was whether to use fixed point or floating point representation in the filter. The paper used floating point, so I kind of assumed we would have to do the same. In addition, I had read that IIR filters are known for having bad numerical stability (because they have feedback, so errors can be compounded and make the response go out of control). These considerations made me assume we would need floating point. But it turned out that the floating point IP was a real pain to work with, and after some more thought I decided floating point would not be necessary for the following reasons. Floating point really shines at representing humongous and teensy numbers at the same time (*e.g.,* 2.3×10^{35} and -8.4×10^{-42}). We didn't need that kind of power for our applications: when I generated the coefficients we would need using Scipy, all of them were within a few orders of magnitude of 1. Thus, I think fixed point was the right representation for our filters. I'm still not sure why the Kugler paper used floating point.

The next step was generating the coefficients for the filters. I had heard MATLAB was really good for this, but I was determined not to break with Python if I didn't have to, and it turned out Scipy (as far as I can tell) is just about as good for this as MATLAB. The real trick was not calculating the right coefficients to use, but actual getting them into our Verilog. The simplest way I found was to write a Python program to generate the IIR coefficients we needed and then plug them into a Verilog coefficient module (similar to the coefficient module for the FIR filter in lab 5a). It may also have been possible to do this with a ROM and .coe file, but this way ended up working out nicely. The Python program took in as parameters the max and min frequencies of the whole bank and the number of subbands to create; this made it extremely easy to change the parameters of the filter bank.

The core implementation of the IIR filter bank was relatively simple. The most interesting problem was how to make the number of subbands configurable in the Verilog. To do this, the `genvar` and `generate` features of Verilog were extremely useful. The `generate` keyword allows you to generate a variable number of modules controlled by a parameter; this allowed us to generate the correct number of instantiations of the coefficient module within the filter bank module (see the Verilog code below; the actual filter bank module obviously has a lot more stuff going on, but the use of `generate` is analogous). It also allows you to change what logic is implemented within a module based on an input parameter; each coefficient module used this feature to instantiate only the logic for the coefficients needed for that specific subband. This allowed us to write a single coefficient module that could be instantiated to supply the coefficients for any subband (see the Verilog code below):

Notional IIR Filter Bank Module with generate statement

```
module iir_filter_bank();
    genvar subband_idx;

    // Each subband needs its own fused-multiply-adder and its own coeff module.
    generate
        for (subband_idx = 0; subband_idx < NUM_SUBBANDS; subband_idx++) begin
            fused_multiply_add my_fma (. . .);

            variable_band_coeffs #(.SUBBAND_IDX(subband_idx))
                iir_coeff_manager (.b_selector(b_selector),
                                   .idx_selector(offset),
                                   .coeff_out(fma_coeff[subband_idx]));
        end
    endgenerate

    always_ff @(posedge clk_in) begin
       // Filter logic
    end
endmodule // fixed_iir_filter
```

IIR Coefficient Module with generate statement

```
module variable_band_coeffs #(
    parameter SUBBAND_IDX = 0);
    generate
        if (SUBBAND_IDX == 0) begin
            always_comb begin
                case ({b_selector, idx_selector})
                    // Coefficient cases
                    . . .
                endcase
            end
        end else if (SUBBAND_IDX == 1) begin
            . . .
        end
    endgenerate
endmodule
```

The problem I spent the most time working on was trying to reduce the number of resources the filter bank took up on the FPGA. The table below shows the bit sizes of each value in the filter:

Value	Number of bits
input	18
coefficients	38
intermediate values	59
output	18

(We may have been able to use less precision, especially for the coefficients—which would have decreased the size of the intermediate values—but I never got around to trying an experiment with fewer bits; I think that resources used would scale linearly with number of bits used, and I hoped I could get better gains in efficiency elsewhere). We were just barely able to fit our design on the board with 32 subbands on both channels (and synthesis on my PC took forever; most was spent in routing—presumably complicated by all the arrays in the code).

My first attempt to improve the size was explicitly to write the multiplications by coefficients as repeated shifted additions. This increased the latency by a factor of 38 (number of shifted additions needed to multiply by a 38-bit coefficient), but latency was almost irrelevant because the filter just needed to finish before the next audio sample came in (over 1000 clock cycles later). I was really surprised when this hardly improved the resource usage at all. I expected that almost all the resources were being used in a gigantic 38- $\times$ 59-bit multiply circuit. Then I made sure that all additions were using the same hardware (I was afraid the 38 shifted additions were using different hardware, which would make the shifted addition about as bad as the multiplication), but this also didn't noticeably help.

From here I think that the biggest expense of each filter was simply routing around the large numbers and arrays. At a high level, I was trying to make each filter component as small as possible (making use of the fact that the delay could be very large); what I should have done was try to reduce the total number of filters instantiated in the first place. Joe made a good point on Piazza that I could probably reuse the same filter hardware multiple times each sample (feeding in different coefficients on each use as appropriate) to get the same effect. If I had more time and ran into more problems with filter bank size, I would try this. One slight snag with this approach is that IIR filters need to store their previous outputs. This is really easy when each filter has its own hardware; if you want to reuse the hardware you would need to feed in not only the coefficients but also the previous outputs.

As illustrated in the block diagram, the filter bank module actually only needs to output the sign bit of the result for each band. This is because the signal goes directly into the zero-crossing spike generator module. I left the output as 18 bits because we still needed to compute the full output (for feedback) and so that we could look at it through the ILA. I also think that Vivado is smart enough to optimize out signals that are never used elsewhere.

2.3 Spike Generation (Matt)

I wrote the spike generation module that turns filter output into rate-coded discrete information. Due to the nature of sound waves, downward zero crossings in an audio signal are relevant markers of activity [7]. In addition, most downstream audio processing in the brain primarily operates on the amplitude modulation and transient rate-coding within a particular frequency channel as determined by the cochlea[8]. In our case, we only focused on transient processing as amplitude is primarily used in calculation of interaural level distances (ILDs), a different feature altogether.

This is the first module in which we are operating on a small input. Our numerical representation of the filter output uses the most-significant-bit position to denote sign, so to catch a downwards zero-crossing we just have to see the MSB go from 0 to 1 (positive to negative). This means that we need to store the previous filter output. The implementation uses a two-stage FSM that is sent into its active or EVAL state on a $0 \rightarrow 1$ in its `valid_in` signal.

It is important to note that in the paper we used as a rough guide for this project[6], the authors utilize a low-pass filter upsampling technique that allows them to run their Spike Generation, Time-Delay Estimator, and Maximum-Finder modules at high clock speeds (200MHz). This was rendered unnecessary by the module inter-operability system we used that cascaded tasks quickly upon operation completion. We note that the authors of the paper used a lot of RAM in their implementation as well – we use none, so that provides another improvement from this module interface design choice.

2.4 Jeffress Circuit (Matt)

From the beginning, the Jeffress circuit module was going to be one we needed to think through carefully. Initially, we were not sure that a Jeffress circuit was the way to go for our time delay extraction or estimation task. However, we eventually found that the G in GCC is big for a reason. Generalizing the math for this type of operation is very resource intensive. The Jeffress circuit is more effective only because it trades full generality for a quantized relationship between its input and output. The quantization performed by the Jeffress circuit geometrically produces a finer grained determination of angle towards the center of the azimuth range (dead-ahead). Therefore, this quantization trade-off is acceptable and even yields further benefits up the processing chain when attention and segregation become involved.

My initial implementation ideas were primarily focused on timing for the coincidence detectors between the two delay lines in the circuit. Research on coincidence detection on FPGAs yielded mostly results that focused on high speed, clockless operation. For example, a paper I looked at was creating coincidence detector latches for experiments with photons[9]. This combined with the emphasis on high clock speeds within the main prior paper led me to believe that we needed a higher level of timing precision than we actually did. It turns out that audio is extremely forgiving when it comes to speed. 62.5kHz provides a lot of time between audio samples. The 65MHz display clock also allows us to take our time. The final latch implementation was a simple AND gate.

The delay lines were also simplified by a relaxing of timing expectations. Propagation along the delay line does not matter when each delay line can only be advanced once every $8\mu s$ due to the way we are sampling audio. This led to the use of arrays as delay lines rather than caring about inter-line propagation which would require some sort of additional specification. We were also worried about spikes missing each other due to shifting behavior. Would they both be advanced at the same time? How do we make the delay lines anti-parallel? How wide are the pulses? Timing relaxation along with the use of the valid in/out scheme solved these problems.

2.5 Rate estimator (Alex):

I wrote the rate estimator module that takes in the coincidence detector output from Jeffress and uses it to keep a running estimate of the rate of fire for each possible delay value. This is a module that we would need multiple copies of in a design where we took advantage of all the subbands (instead of only using 1, as we did in the video demo). One simple way of doing this would be to simply sum up the number of times each detector fired over some period (say, 1000 cycles), find the maximum and output it, and then repeat. Honestly, I don't know how this would have behaved compared to my implementation. Our rate detector did keep a counter for each possible delay, but instead of resetting it every 1000 cycles it multiplied it by a decay factor $\alpha < 1$ (which was always a fraction over a power of 2 to make the arithmetic easy). Theoretically this kept an actual running rate estimate available at each cycle instead of only every 1000 cycles. Initially my decay rate was too slow and my period was too long, so this made the detector very slow to respond to changes in source location. To tune the parameters I set them based on the switches (as they were in the actual demo); I found that incrementing the counter by 3 every time a detector fired and applying a decay of $\alpha = \frac{63}{128}$ (now I realize that that's basically $\frac{1}{2}$, but when I was tuning it with the switches I didn't realize that) every 1000 cycles gave the best results for our demo. When the tone generator was not running the rates almost never exceeded the threshold (*i.e.,* we very rarely got a false detection), and with the tone generator running it would almost always detect it.

We were hoping to be able to detect the location of human speech, but this didn't work with the parameters I set in the final demo. Now that I think about it, I wonder if a slightly less aggressive decay rate could have solved that problem. A fundamental challenge with locating speech is that it's not continuous like the tone we used in the demo; it does seem, however, like the circuit clock is fast enough that it should be able to detect even very short sounds (like single words).

2.6 Display (Alex):

The final module was the display output module. The display had 3 major components (see figure 7):

1. The bottom row where we highlighted the block with the maximum rate (if it exceeded the threshold).
2. The middle section that displayed the current rate estimate for that detector as the height of a bar (like a histogram) and showed the threshold.
3. The top semicircle that showed the calculated angle of arrival if a rate exceeded the threshold.

The first 2 were relatively simple to implement. The bottom bar was simply a matter of calculating the index of the coincidence detector corresponding to the current `hcount` and drawing a color if it was equal to the current delay estimate and if the max rate was over the threshold. The middle section was basically the same, except we drew a color iff the rate value at the correct index was greater than our current height.

What I think is really cool about the semicircular display is how natural it is to the time-delay-location algorithm. If your mics are on the horizontal axis and centered on the origin, and if you know that the sound source is located on the semicircular arc, then after computing the delay (as it is shown in the bottom bar of the display) the source must be on the section of the semicircle directly above the highlighted block in the bottom bar. This follows from the formula

$$\cos\theta = k \cdot \frac{c}{L \cdot f_s}$$

To actually draw the display, we just need to decide whether the current point (x, y) that `hcount, vcount` represents is inside the "pie-slice" of the circle that extends to the appropriate piece of the arc. Consulting figure 2.6, the nicest way to express this algebraically is that we need the slope $\frac{y}{x}$ to lie between the slopes

$$\frac{y_L}{x_L} \quad \text{and} \quad \frac{y_R}{x_R}$$

The "between" relation initially seemed like it would be a pain, but then I realized that you can just compare the slope to both with $\leq$ and then XOR the results. But there's still a problem because division is expensive in hardware; the best way to get around this is to check the cross-multiplication

$$(x \cdot y_L \leq y \cdot x_L) \oplus (x \cdot y_R \leq y \cdot x_R)$$

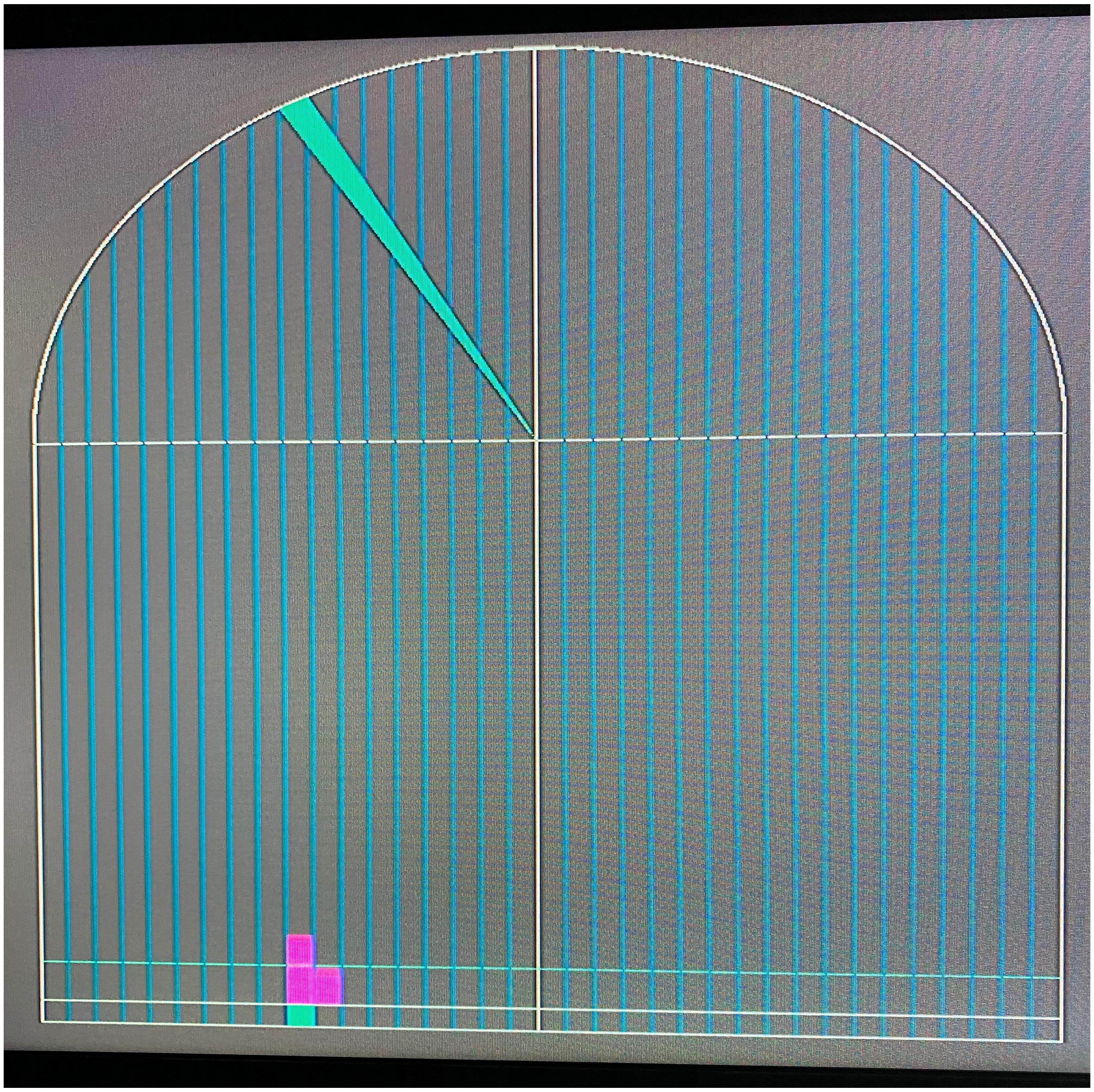

Figure 7: A picture of the display in the final demo.

(I should really be more careful with the signs and directions of inequalities, but we will later square everything; we need an additional sign check to account for this) This is still problematic because the values y_L and y_R involve a square root (we can easily calculate x_L and x_R from the delay value). We could compute

$$y_L = \sqrt{R^2 - x_L^2} \qquad y_R = \sqrt{R^2 - x_R^2}$$

but square roots are really expensive in hardware. Better to square both sides of each inequality and check

$$(x^2 \cdot y_L^2 \leq y^2 \cdot x_L^2) \oplus (x^2 \cdot y_R^2 \leq y^2 \cdot x_R^2)$$

This is perfect, except that we lose sign information about x, x_L, and x_R (all the y values are always positive; if we check only these inequalities, then we'll end up drawing source color at both x and $-x$, which is clearly

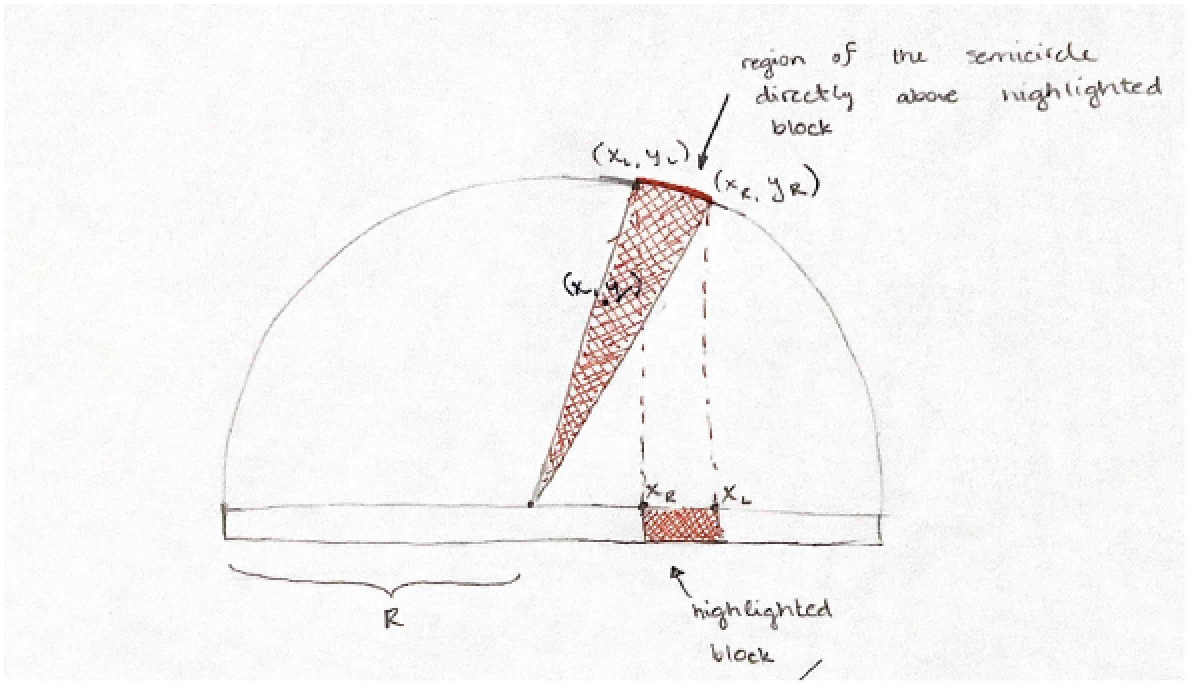

Figure 8: A sketch of the semicircular display. The section of the circle directly above the highlighted block represents the set of possible source locations on the semicircle.

not right). The best way I found to solve the sign ambiguity problem is just to do a simple check of the MSB of x with the MSBs of x_L and x_R. This computation with all the squaring took an additional 2 cycles of pipelining (the rest of the display modules didn't require any pipelining). This was a bit of a pain and created some bugs (for example, at one point I was not passing along the sign check of x correctly and this caused minor issues and took a little while to find).

To debug the display, I didn't use mainly software test-benches (except to make sure that the pipelining and arithmetic for the semicircular display was working) like I did for the rest of my modules. I would just compile the demo and set the inputs to the display (*i.e.,* the rates and delay, which are normally output from the Jeffress module) from the switches on the board. Vivado would optimize out the whole audio/Jeffress pipeline (because its output was never used anywhere), so the synthesis would go really fast; then I could just test the display with different delay and rate values. This helped get the display working quickly at the end of the project.

I found another little trick with the graphics that I thought was cool. I actually built the rate-height-histogram and semicircle display modules independently. For the demo I thought I could just bitwise OR the resulting pixel values together (once I pipelined the rate-height-histogram pixel value—otherwise the rate bars would be slightly offset), but then I realized that this would not work if the background color was not black (and I had been trying to come up with a nice color scheme). Then I realized that, as long as both modules used the same background color and never tried to write to the same pixel (*i.e.,* at least one of them always output the background color), I could synthesize the output pixel colors by bitwise XORing them together and then XORing the result with the background color. The 3 things that make this trick work are

1. Bitwise XOR is associative (this follows from the fact that for single bits, $a \oplus b$ is the LSB of $a + b$).

2. $a \oplus a = 0$ for all a.

3. $a \oplus 0 = a$ for all a.

Say that the two display modules output pixel values p and q, and that B is the common background color. Then the claim is that

$$p \oplus q \oplus B$$

gives the desired color (as long as either p or q is equal to B). Assume for the sake of argument that p is the useful color and $q = B$. Then

$$p \oplus q \oplus B = p \oplus (q \oplus B) = q \oplus (B \oplus B) = q \oplus 0 = q$$

I thought this was a nifty trick and could be more useful for other applications where you really don't want a black background.

3 Conclusion

3.1 Future Improvements

There are a number of obvious next steps for the project:

3.1.1 Combine Delay Information from Different Subbands

In the current demo there is only a single copy of the Jeffress hardware, and it computes on whichever subband is MUXed into it. All of the display information is thus based on only a small frequency range of the audio input. If we had more time we would create Jeffress circuits and rate-estimator circuits to operate on each subband in parallel. Then we would get an estimated delay (and know whether the maximum rate exceeded the threshold) for each subband; to get the best overall delay estimate we could average the delay estimates over all channels whose max rate exceeded the threshold (we could also include something to discard outliers). This is basically the approach taken in the Kugler paper.

It is interesting to consider how much we would gain by combining information from multiple subbands. It would allow us to perform the task on not just tones but speech or music. Displaying the information of that more complex signal would therefore also be very worthwhile.

3.1.2 Include More Microphones

We had hoped to use data from 3 mics to accomplish at least 2-D localization, but we ended up using only 2 microphones for the sake of time. In theory, 3 mics would not require much extra work; we would just need to add a third filter bank and then another set of Jeffress circuits and rate estimators. One of the main reasons we chose not to worry about that was because we didn't have a compelling way to display 2-D location information.

3.1.3 Additional Processing Layers

One of the techniques mentioned in the Kugler paper is a spectral masking step in which the behavior of the various frequency subbands is analyzed to determine which actually have useful localization information. This takes the form of a set of leaky integrators that both approximate a coherence-like measure between the signals and threshold the latch output based on their most recent behavior.

Another layer that would be cool would be a 3D version of spectral masking or just some sort of 3D interpreter. Source segregation and echo suppression could also start to be looked at with this 3D analysis.

3.1.4 Improve the Filter Bank Module

As noted earlier, the filter bank module could probably be improved significantly. Following up on Joe's recommendation, I would like to implement each filter bank with a single IIR filter instantiation and reuse it for each subband (making use of the fact that we have 1600 100 MHz cycles to work with between each audio sample taken at 62.5 KHz). It would also be interesting to see if there is a smarter way to implement the filters overall.

3.2 Acknowledgements

We'd like to thank our 6.111 advisor Tomas Palacios for his guidance and advice throughout the project. Brian Sennett, our industry mentor from Bose, offered an audio expert's perspective and helped us prioritize features for the final demo. Joe Steinmeyer, Gim Hom, and Keshav Gupta all provided critical help over Piazza. Thanks to Josh McDermott for the introduction to Jeff.

References

[1] C. T. Ishi, J. Even, and N. Hagita, "Using multiple microphone arrays and reflections for 3D localization of sound sources," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3937–3942, Nov. 2013. ISSN: 2153-0866.

[2] C. Liu, B. C. Wheeler, W. D. O'Brien, R. C. Bilger, C. R. Lansing, and A. S. Feng, "Localization of multiple sound sources with two microphones," *The Journal of the Acoustical Society of America*, vol. 108, pp. 1888–1905, Sept. 2000. Publisher: Acoustical Society of America.

[3] N. Sakamoto, W. Kobayashi, T. Onoye, and I. Shirakawa, "DSP implementation of 3D sound localization algorithm for monaural sound source," in *ICECS 2001. 8th IEEE International Conference on Electronics, Circuits and Systems (Cat. No.01EX483)*, vol. 2, pp. 1061–1064 vol.2, Sept. 2001.

[4] L. A. Jeffress, "A place theory of sound localization.," *Journal of Comparative and Physiological Psychology*, vol. 41, no. 1, pp. 35–39, 1948.

[5] P. Joris, P. Smith, and T. Yin, "Coincidence Detection in the Auditory System," *Neuron*, vol. 21, pp. 1235–8, Jan. 1999.

[6] M. Kugler, T. TOSSAVAINEN, S. KUROYANAGI, and A. IWATA, “Design of a Compact Sound Localization Device on a Stand-Alone FPGA-Based Platform,” *IEICE Transactions on Information and Systems*, vol. E99.D, pp. 2682–2693, Nov. 2016.

[7] Y.-I. Kim and R. Kil, “Estimation of Interaural Time Differences Based on Zero-Crossings in Noisy Multisource Environments,” *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 15, pp. 734–743, Mar. 2007.

[8] A. J. E. Kell, D. L. K. Yamins, E. N. Shook, S. V. Norman-Haignere, and J. H. McDermott, “A Task-Optimized Neural Network Replicates Human Auditory Behavior, Predicts Brain Responses, and Reveals a Cortical Processing Hierarchy,” *Neuron*, vol. 98, pp. 630–644.e16, May 2018.

[9] R. Joost and R. Salomon, “CDL, a Precise, Low-Cost Coincidence Detector Latch,” *Electronics*, vol. 4, pp. 1018–1032, Dec. 2015.